Single Responsibility Principle (SRP)

import java.util.ArrayList;

public class Customer {

private String name;

private int age;

private ArrayList<Item> listsOfItems;

public Customer(String name, int age) {

this.name = name;

this.age = age;

this.listsOfItems = new ArrayList<Item>();

}

public void addItem(Item item) {

this.listsOfItems.add(item);

}

public String getName() {

return name;

}

public int getAge() {

return age;

}

public ArrayList<Item> getListsOfItems() {

return listsOfItems;

}

public static void main(String[] args) {

ArrayList<Item> items = new ArrayList<Item>();

items.add(new Item(25));

items.add(new Item(45));

Customer fred = new Customer("Fred", 25);

fred.addItem(new Item(25));

fred.addItem(new Item(45));

long bill = BillCalculator.*calculateBill*(fred.getListsOfItems(), 10);

ReportGenerator.*generateReport*("CSV", fred, bill);

Customer karen = new Customer("Karen", 30);

karen.addItem(new Item(10));

karen.addItem(new Item(10));

bill = BillCalculator.*calculateBill*(karen.getListsOfItems(), 20);

ReportGenerator.*generateReport*("XML", karen, bill);

}

}

import java.util.ArrayList;

public class BillCalculator {

public static long calculateBill(ArrayList<Item> items, long tax) {

long bill = 0;

for (Item item : items) {

bill += item.getPrice();

}

return bill + tax;

}

}

public class ReportGenerator {

public static void generateReport(String reportType, Customer customer, long bill) {

if (reportType.equalsIgnoreCase("CSV")) {

System.***out***.println("CSV Report: " + customer.getName() + "'s bill is " + bill);

}

if (reportType.equalsIgnoreCase("XML")) {

System.***out***.println("XML Report: " + customer.getName() + "'s bill is " + bill);

}

}

}

The Item class I left unchanged; therefore I did not include it in this report.

Q1 – In this refactored code, there are two new classes, BillCalculator and ReportGenerator, with a single responsibility for each. The Customer class is now responsible only for storing the customer's personal information and the items they have bought.

In the main method, there’s a Customer objects and can populate their listsOfItems fields. Then, pass the listsOfItems and tax amount to BillCalculator to get the total bill, and then pass the reportType, Customer object, and bill amount to ReportGenerator to generate the report.

Q2 – One thing that violates SRP is by having multiple responsibilities in the same method. For example, there’s a method that is responsible for parsing the CSV file, validating the data, initializing the board cells, creating and updating several maps, and populating the room and cell data structures. These responsibilities can be separated into smaller, more focused methods, each with a single responsibility. We can refactor the code by extracting several smaller methods that have a clear and single responsibility. By refactoring the code into smaller methods with clear responsibilities, it makes it easier to test the code, as we can test each method in isolation, instead of having to test the entire loadLayoutConfig method.

In the Game class, it takes on multiple responsibilities such as handling the game logic, drawing the game board and players on the GUI, and managing the game state. This violates SRP because a class should have only one reason to change, but in this case, any change in the GUI or game logic would require modifications to the Game class. To solve this violation, we could divide the responsibilities into separate classes. We could create a separate class to handle the game logic, such as GameLogic, which would contain methods for checking the accusation and handling the suggestion. This would make the code more modular and easier to maintain.

The Board class also violates SRP because it contains methods that have multiple responsibilities. For example, it has a MouseListener method and an ActionListener method that handle different types of events, and a paintComponent() method that handles both board drawing and card drawing. To solve this, we could create separate classes or interfaces for each type of event handling, and separate methods for each type of drawing. For example, we could create a MouseListenerManager and ActionListenerManager class that handle mouse and action events, and separate paintComponent() methods for board drawing and card drawing.

Open-Closed Principal (OCP)

public class Pet {

public String petName;

public *PetType* petType;

public Pet(String petName, *PetType* petType) {

this.petName = petName;

this.petType = petType;

}

public String getPetName() {

return petName;

}

public *PetType* getPetType() {

return petType;

}

*@Override*

public String toString() {

return petName + " - " + petType.toString();

}

}

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

public class PetStore {

private List<Pet> pets = new ArrayList<>();

private Map<PetType, PetAction> petActions = new HashMap<>();

public PetStore() {

petActions.put(*PetType*.***CAT***, new CatAction());

petActions.put(*PetType*.***DOG***, new DogAction());

}

public void performAction(Pet pet) {

PetAction action = petActions.get(pet.getPetType());

action.performAction(pet);

}

public void addPet(Pet pet) {

pets.add(pet);

}

public void listPets() {

for(Pet pet : pets) {

System.***out***.println(pet);

}

System.***out***.println();

}

public static void main(String[] args) {

PetStore myStore = new PetStore();

myStore.addPet(new Pet("Buttons", *PetType*.***CAT*** ));

myStore.addPet(new Pet("Boxer", *PetType*.***DOG*** ));

myStore.addPet(new Pet("Dax", *PetType*.***CAT*** ));

myStore.addPet(new Pet("Spot", *PetType*.***DOG*** ));

myStore.listPets();

for(Pet pet : myStore.pets) {

myStore.performAction(pet);

}

}

private interface PetAction {

public void performAction(Pet pet);

}

private class CatAction implements PetAction {

public void performAction(Pet pet) {

System.***out***.println("Watching " + pet.getPetName() + " sleep");

System.***out***.println("Giving " + pet.getPetName() + " some catnip");

System.***out***.println("Hearing meow from " + pet.getPetName());

System.***out***.println();

}

}

private class DogAction implements PetAction {

public void performAction(Pet pet) {

System.***out***.println("Throwing a frisbee to " + pet.getPetName());

System.***out***.println("Giving " + pet.getPetName() + " a bone");

System.***out***.println("Hearing woof from " + pet.getPetName());

System.***out***.println();

}

}

}

public interface PetAction {

void performAction();

}

public class CatAction implements PetAction {

private String petName;

public CatAction(String petName) {

this.petName = petName;

}

public void performAction() {

System.***out***.println("Watch " + petName + " sleep");

System.***out***.println("Give " + petName + " some catnip");

System.***out***.println("Meow");

System.***out***.println();

}

}

public class DogAction implements PetAction{

private String petName;

public DogAction(String petName) {

this.petName = petName;

}

public void performAction() {

System.***out***.println("Throw a frisbee to " + petName);

System.***out***.println("Give " + petName + " a bone");

System.***out***.println("Woof");

System.***out***.println();

}

}

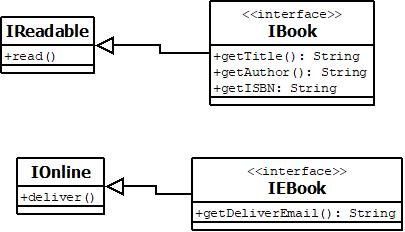
The original code violated the Open/Closed Principle (OCP) because it was not closed for modification, since any new behavior or pet type required modifying the existing code, which lead to potential issues. The original code used if-else conditions to determine the type of pet and perform the corresponding action, which made the code less maintainable and less extensible.

The new code adheres to OCP by using polymorphism and inheritance to extend the functionality of the PetStore without modifying its existing code. The code defines a new abstract class called PetAction that encapsulates the behavior of different actions that can be performed on pets. Each pet action is implemented as a subclass of PetAction, and it overrides the performAction() method to implement its specific behavior. By doing this, the code makes it easy to add new pet actions without modifying the existing code.

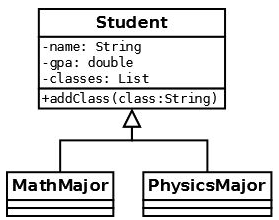
Interface Segregation Principle (ISP)

This design violates the Interface Segregation Principle (ISP) because the EBook class inherits from the book interface, but it does not need or use all the methods defined in the book interface. Specifically, the EBook class has an additional attribute called deliveryEmail that is not relevant or useful to the book interface. By inheriting from the book interface, the EBook class exposes all the public methods defined in the book interface, including those that are not applicable to the EBook class.

According to the ISP, clients should not be forced to depend on methods they do not use. In this case, the EBook class is a client of the book interface, but it is being forced to depend on methods it does not use or need, violating the ISP. To adhere to the ISP, the EBook class should only inherit from an interface or abstract class that defines methods that are applicable and relevant to the EBook class.



**BONUS QUESTION**



We do not need the two child classes MathMajor and PhysicsMajor in this design. The presence of these child classes violates the Single Responsibility Principle (SRP) because it assumes that all math majors will have a certain behavior and all physics majors will have another behavior. However, this assumption is not always true.

A better approach would be to have a single Student class and use composition to represent the majors. We can introduce a Major interface, which can have multiple implementations, such as MathMajor and PhysicsMajor, and add a major field of type Major to the Student class. This way, the Student class can have any major, not just limited to math or physics.